Decrypting Substitution Ciphers with Genetic Algorithms

Jason Brownbridge
Department of Computer Science
University of Cape Town
jbrownbridge@gmail.com
17/03/2007

In this paper we explore the use of Genetic Algorithms to break a Substitution Cipher which forms part of the Monoalphebetic cipher family. We created a hybrid solution using several cryptographic techniques such as frequency analysis in order to produce a fitness function that resulted in rapid convergence.

Table of Contents

1. Introduction 2
2. Substitution Cipher 2
3. Genetic Algorithm 3
   3.1. Encoding 3
   3.2. Selection 3
   3.3. Fitness 4
   3.4. Crossover 5
   3.5. Mutation 5
   3.6. Elitism 6
   3.7. Implementation 6
4. Experimentation 7
   4.1. Population Size 8
   4.2. Tournament Size 8
   4.3. Probability Crossover 9
   4.4. Probability Mutation 9
   4.5. Percentage Elitism 10
   4.6. Population Seeding 10
5. Conclusions 11
6. Future Works 11
References 12

1. Introduction

In this paper we apply the technique of genetic algorithms to the problem of finding the key for a particular Substitution Cipher.

Since Genetic Algorithms are primarily used to efficiently search a large problem space we thought they would be ideally suited for searching the large key space.

Firstly we frame the problem in terms of defining what a substitution cipher is and then which variety of substitution ciphers we are concerned with in this paper.

Then we discuss several design decisions we made in the implementation of our genetic algorithm taking into account the domain knowledge for this particular problem.

Lastly we conduct several experiments by varying the parameters of our genetic algorithm and seeing what impact they have on the accuracy and speed of the algorithm.

2. Substitution Cipher

A cipher is an algorithm for encrypting plain text into cipher text and vice versa. The substitution cipher replaces every instance of a particular letter in the plain text with a different letter from the cipher text.

Thus a substitution cipher key can be defined as the set of one-to-one mappings relating every letter in the plain text alphabet with the corresponding letter in cipher text alphabet.

Such a key is normally defined using a table and an example key is included below.

TABLE 1: EXAMPLE SUBSTITUTION CIPHER KEY

Alphabet	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Key	E	U	S	X	V	A	R	T	I	K	C	Z	M	G	Q	Y	H	O	J	P	W	D	B	N	L	F

When using the key above to encrypt a message each letter in the Alphabet row is replaced by the letter in the Key row below it, when decrypting the reverse occurs.

Thus the "*THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG*" can be encrypted to *"PTV HWISC UOQBG AQN KWMYJ QDVO PTV ZEFL XQR"*.

On the surface this cipher seems to be a strong one since there are 26 possibilities to choose from for the first letter, 25 for the second, 24 for the third and so on. One can clearly see that there are in fact 26! possible keys.

$$26! = 26 * 25 * 24 * \dots * 1 \cong 4.03^{26}$$

However this cipher is particularly vulnerable to a technique known as frequency analysis since although it does change the letters in the plain text to different ones in the cipher text it does not change the underlying frequency of those letters.

Thus by comparing the frequencies of letters in the cipher text to a table of known letter frequencies for the plain text language the key space can be reduced drastically. Furthermore you can also group letters into n-grams where *n* represents the number of letters in the n-gram and look up the corresponding frequencies for those.

The substitution ciphers we will be examining in this paper consist of the set of substitution ciphers which can be defined by the alphabet of uppercase letters A-Z. For simplification sake we will assume the plain text consists of only upper case letters and punctuation but that the punctuation will not be encrypted.

3. Genetic Algorithm

The problem we are attempting to solve with this genetic algorithm is that of finding the correct key from all 26! possible keys that will enable us to decrypt the cipher text into plain text.

We attempt to make use of our domain knowledge about the nature of substitution ciphers and their weakness to frequency analysis techniques in order to improve our fitness function in terms of rate of convergence as well as probability of convergence.

Genetic Algorithms offer several advantages over traditional frequency analysis methods as they require no human intervention unless they fail to converge. However this can be prevented by running the algorithm for fixed maximum number of generations as well as running it several times and comparing the results in order to increase the probability of convergence.

We will now discuss our design decisions in terms of encoding, training, fitness, selection, crossover and mutation for our implementation of this hybrid genetic algorithm.

3.1. Encoding

Each individual in the population has one chromosome consisting of only 26 characters where each character is a different letter from the English alphabet.

You can then think of this chromosome as consisting of 26 genes where each gene is defined by the character as well as its position in the string.

Thus the chromosome "ZYXWVUTSRQPONMLKJIHGFEDCBA" represents a key which would map every letter "A", "Y" and "X" in the plain text to the letters "Z", "Y" and "X" in the cipher text and so on. Where (0, "Z"), (1, "Y") and (2, "X") can be thought of as the genes.

3.2. Selection

Selection is done by a tournament process [1] where the tournament is held between arbitrary numbers of individuals.

Each individual is chosen at random from the environment and the winner is selected according to the following method (where p is one of the parameters that must be defined for the genetic algorithm):

- The best (highest fitness) individual is selected as the winner with some probability p
- The second best individual is selected as the winner with some probability $p * (1 – p)$
- The third best individual is selected as the winner with some probability $p * (1 – p) * (1 – p)$
- And so on.

Since each child requires two parents we must hold at least two tournaments, one to determine each parent. This imposes certain limitations on the tournament members as we have to ensure each tournament consists of distinct individuals and that the intersection of the individuals participating in each tournament is the empty set. We have to do this to ensure that both parents are distinct individuals as it would be problematic if the two individuals selected were in fact the same individual.

Thus in order to produce k children we have to hold at least $\frac{k}{2}$ tournaments where the last child is discarded if k is an odd number.

3.3. Fitness

In order to understand our fitness function we must first define what an n-Gram is. An *n*-Gram is a group of *n* letters which appear consecutively within a word.

The algorithm for finding all the *n*-Grams in a word can be defined as follows:

- Divide the word up using two pivots with both initially located before the first letter.
- Advance the second pivot *n* characters forward if possible, if not no *n*-Grams exist so quit.
- The letters between the two pivots will now give you your first *n-Gram.*
- In order to find the remaining *n*-Grams advance each pivot one character forward at a time. Each time you are able to advance the two pivots the characters between them define another *n*-Gram.
- When you are no longer able to advance the two pivots you have found all the *n*-Grams so quit.

An example follows below:

If $n = 3$

||COLLETION – Initial state both pivots are located before the first character.

|COL|LETION – First n-Gram found: COL

C|OLE|CTION – Second n-Gram found: OLE

CO|LEC|TION – Third n-Gram found: LECT

And so on.

Thus the word "COLLECTION" consists of the following n-Grams:

TABLE 2: TABLE OF N-GRAMS FOR N=2 AND N=3

$n = 2$	$n = 3$
CO	COL
OL	OLL
LL	LLE
LE	LEC
EC	ECT
CT	CTI
TI	TIO
IO	ION
ON	

The reason *n*-Grams are so important is because their frequency for particular language can be determined by using a set of training texts and counting the occurrences of each *n*-Gram within the texts.

The frequency of the *n*-Grams within a large enough sample of cipher text will be similar to the frequency deduced from the training texts.

Thus we determine the fitness of particular individual as follows:

- Let *n* for all *n*-Grams be 3
- Let Frequency be defined as the number of occurrences of a n-Gram particular
- Let T be the set of n-Grams found within the training text
- Let N(x) be the set of n-Grams found within the text decrypted by the chromosome of the individual x
- $\text{Let } F_T(y) = \begin{cases} \text{Frequency of y within training text if } y \in T \\ 0 \text{ otherwise} \end{cases}$
- $\text{Let } F_P(x, y) = \begin{cases} \text{Frequency of y within text decrypted by indiviual x if } y \in N(x) \\ 0 \text{ otherwise} \end{cases}$
- $\text{Fitness}(x) = \sum_{y \in N(x)} F_P(x, y) \times \log_2 F_T(y)$

In simpler terms the fitness of a particular individual is merely the sum of all the logarithms base 2 of the frequency of the *n*-Grams which appear in the plain text generated by decrypting the cipher text using the chromosome of the individual as the decryption key.

We used the logarithm function to deal with sparse distributions of *n*-Gram frequencies where *n*-Grams with higher frequencies would completely annihilate the *n*-Grams with lower frequencies.

We found the logarithm with base 2 to work well as our smoothing function. Lastly we choose *n* to be 3 for our *n*-Grams as it produced the best results.

3.4. Crossover

After examining several possible choices for crossover operators we decided to use the Position-Based Crossover (PBX) operator [2].

Position-Based Crossover can be defined as follows:

- Given the two parents produced by the selection process.
- Choose up to *k* points of crossover.
- Copy the values at those *k* points from the first parent into the same *k* points in the first child.
- Then iterate through the remaining values from the second parent filling in the missing values in the first child.
- To generate the second child you follow the same process except you reverse the parents.

As you can see this crossover operator is used mainly when you have a set of distinct genes and the genetic information you wish to pass to children is in fact the position of those genes within the chromosome.

This ideally suited for our problem where the genes can be thought of as consisting of a position as well as distinct character from a fixed set of characters namely the uppercase letters from the English alphabet.

3.5. Mutation

We decided to implement mutation using the Swap Mutation operator [2]. This operator works by selecting two random (but different) positions within the chromosome and then swapping the letters each position with each other.

We felt this was an excellent choice as it would allow us to avoid local maxima which would occur when the majority of letters with high frequencies were in the correct positions and any crossover operation would result in a decrease in fitness for that individual.

3.6. Elitism

Since we use a tournament selection process we have no guarantee that the fittest individuals from each generation will actually produce the offspring for the next generation. This in fact would be largely dependent on the tournament size as well as the probability that the best individual in the tournament will be selected as the winner.

Thus to ensure the survival of individuals with high fitness values and to increase the average fitness of the each generation we use Elitism to copy a fixed percentage of the fittest individuals from the current population into the next generation.

3.7. Implementation

We implemented our solution using C# and created a graphical user interface in order to experiment with adjusting the various parameters for the genetic algorithm. An example screenshot of our interface is included below.

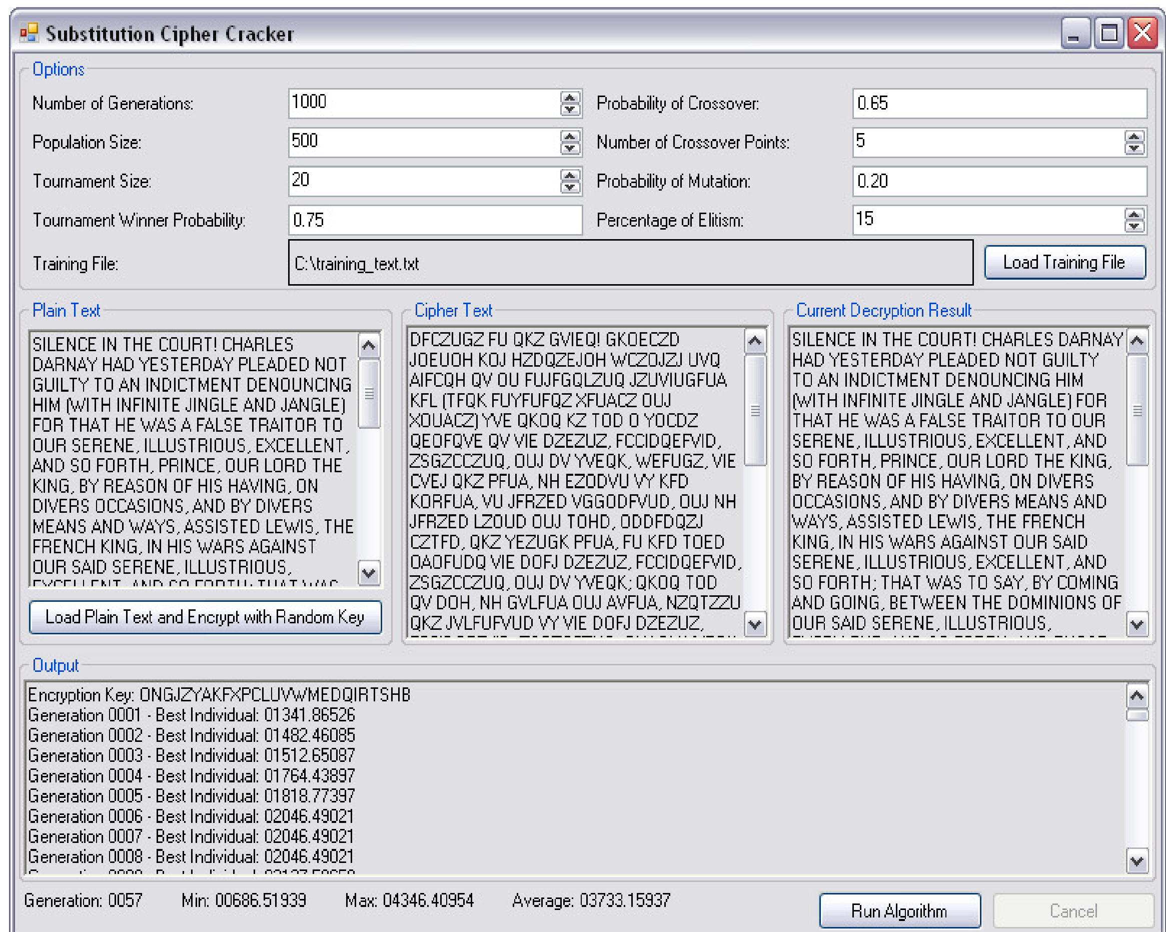

FIGURE 1: SCREENSHOT OF USER INTERFACE AFTER SUCCESSFUL DECRYPTION

4. Experimentation

We conducted all our experiments using the user interface mentioned in the previous section. The interface generates a comma separated list of data where each comma represents a column and each new line represents a row list of data. The columns consisted of the generation number, minimum fitness, maximum fitness and average fitness and where the rows represent the values for these columns for each generation.

All graphs shown were generated by importing the comma separated list of data into Microsoft Excel 2007 and then using it to generate the graphs.

We used the following default set of default parameters (varying only a single parameter for each experiment):

- Maximum Number of Generations: 1000
- Population Size: 500
- Tournament Size: 20
- Tournament Winner Probability: 0.75
- Probability of Crossover: 0.65
- Number of Crossover Points: 5
- Probability of Mutation: 0.20
- Percentage of Elitism: 15%
- Population Seeding: TRUE

For training the fitness function we used "A Tale of Two Cities" by Charles Dickens which was provided by Project Gutenberg.

The cipher text used was an extract from the novel mentioned above and was encrypted using a random encryption key for each run. The original plain text was:

> *"Silence in the court! Charles Darnay had yesterday pleaded Not Guilty to an indictment denouncing him (with infinite jingle and jangle) for that he was a false traitor to our serene, illustrious, excellent, and so forth, prince, our Lord the King, by reason of his having, on divers occasions, and by divers means and ways, assisted Lewis, the French King, in his wars against our said serene, illustrious, excellent, and so forth; that was to say, by coming and going, between the dominions of our said serene, illustrious, excellent, and so forth, and those of the said French Lewis, and wickedly, falsely, traitorously, and otherwise evil-adverbiously, revealing to the said French Lewis what forces our said serene, illustrious, excellent, and so forth, had in preparation to send to Canada and North America. This much, Jerry, with his head becoming more and more spiky as the law terms bristled it, made out with huge satisfaction, and so arrived circuitously at the understanding that the aforesaid, and over and over again aforesaid, Charles Darnay, stood there before him upon his trial; that the jury were swearing in; and that Mr. Attorney-General was making ready to speak."*

4.1. Population Size

This parameter defines the population size for the algorithm. We varied the population size between 200, 500 and 1000 individuals to see how it affected convergence. Both the 500 and 1000 sized populations converged within 80 steps; however the first one did converge slightly faster. One of the advantages of a population with a smaller size is that the actual execution time for the algorithm decreases due to the reduction in the computation load. However as seen in *Figure 2 below* if you decrease the population size too much you will not have enough diversity in the system to reach a solution.

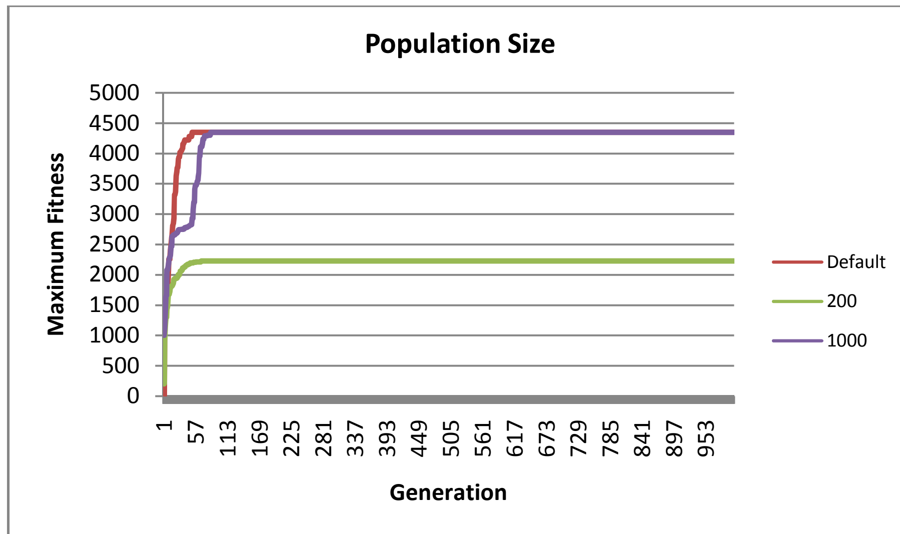

FIGURE 2: VARYING POPULATION SIZE

4.2. Tournament Size

This parameter determines the number of individuals from the population which participate in each tournament. We varied the tournament size between 15, 20 and 25 with similar results as seen in *Figure 3* below. However larger tournaments seemed to converge slightly faster.

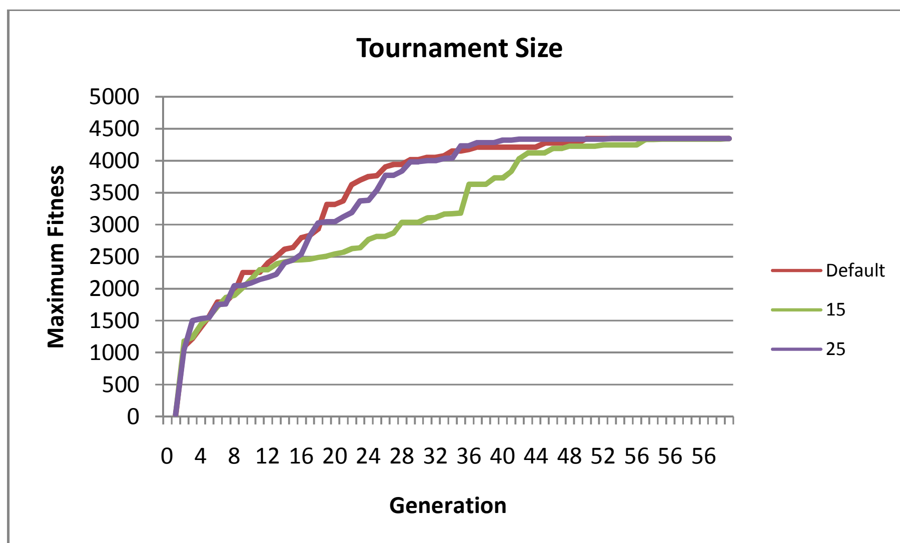

FIGURE 3: VARYING TOURNAMENT SIZE

4.3. Probability Crossover

This parameter determines the probability of crossover occurring, if crossover does not occur then each child produced is merely a copy of the corresponding parent. Crossover occurs before mutation so the final version of the child may be slightly different. We varied the probability of crossover occurring between 0.55, 0.65 and 0.75. When using both the 0.65 and 0.75 we converged to a solution with 56 steps, however the lower value seemed to converge slightly faster as seen in *Figure 4* below. However when we decreased the probability to 0.55 it did not converge. One can assume this was because there was not enough genetic diversity being generated by the crossover operator.

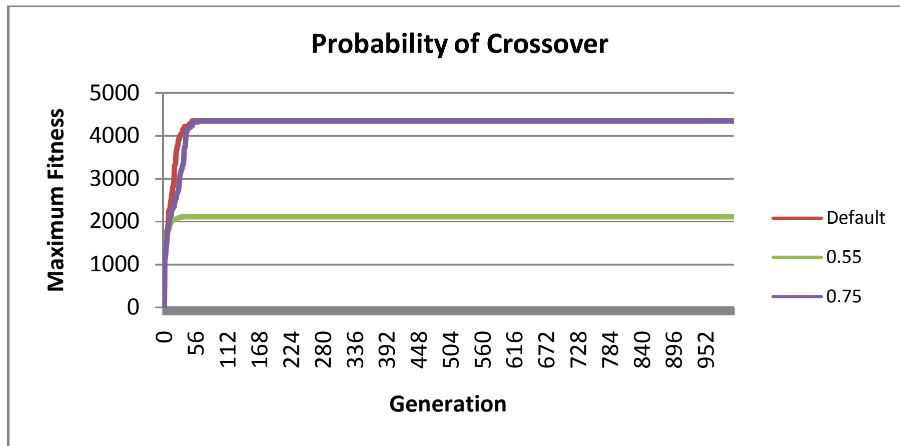

FIGURE 4: VARYING PROBABILITY OF CROSSOVER

4.4. Probability Mutation

This parameter determines the probability of mutation occurring after crossover takes place. We varied the probability of mutation between 0.1, 0.2 and 0.3. All parameters converged to solution in at most 53 steps. We did notice however that the higher mutation probabilities tended to converge more rapidly as seen in *Figure 5* below. This seems logical as this particular problem is highly sensitive to local maxima and a higher mutation rate will help mitigate against this.

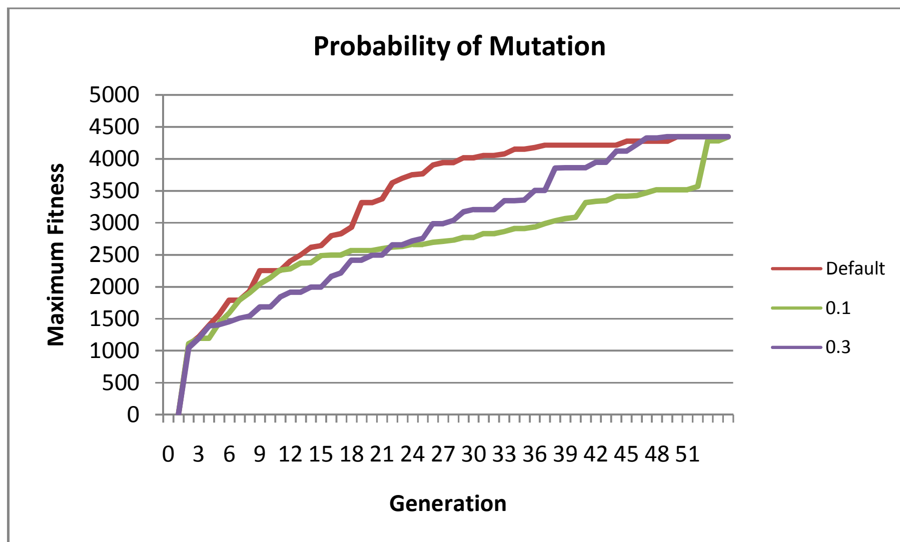

FIGURE 5: VARYING PROBABILITY OF MUTATION

4.5. Percentage Elitism

This parameter determines which percentage of the fittest individuals from the current generation is copied into the next generation. We varied the percentage between 10, 15 and 20 percent. Both 15 and 20 percent elitism resulted in convergence within 56 steps and however 10 percent elitism did not result in convergence as seen in *Figure 6* below.

Since we base selection of individuals for procreation based on tournament selection we have no guarantee the fittest individuals will procreate as that largely depends on the tournament size. Thus by increasing the percentage of elitism we ensure individuals with fitness survive for longer and have a greater chance of passing their genetic material on to subsequent generations.

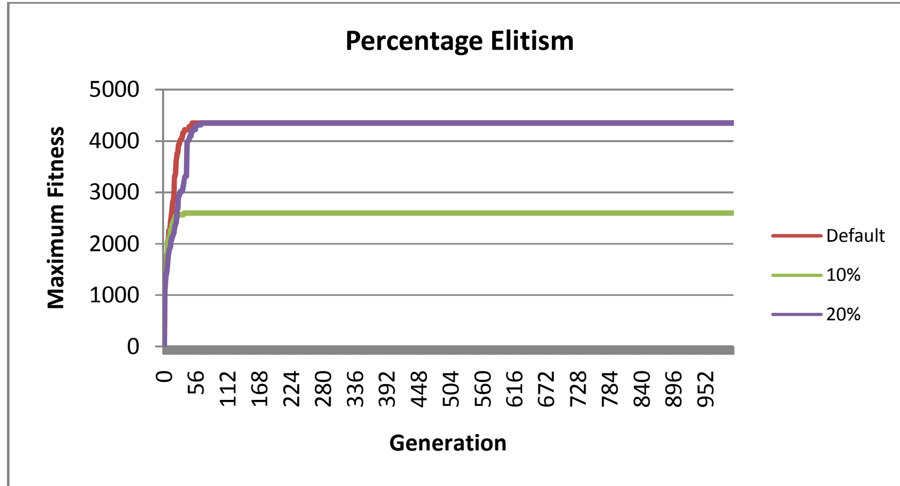

FIGURE 6: VARYING PERCENTAGE OF ELITISM

4.6. Population Seeding

This parameter determines whether a percentage of the initial population is generated by examining the frequency of letters from the training text against the frequency of letters from the cipher text and associating them with a certain probability.

As seen below in *Figure 7* seeding the populate results in a somewhat faster convergence rate. However we were surprised with these results as we expected a much larger boost in initial fitness.

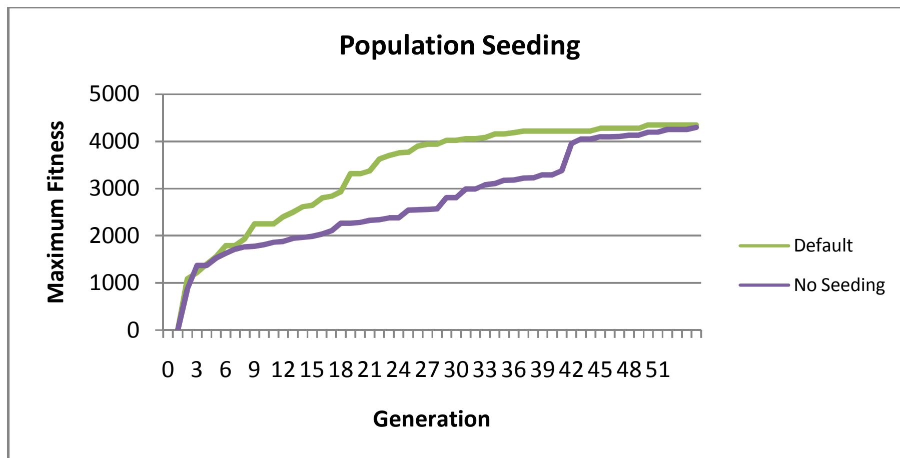

FIGURE 7: VARYING POPULATION SEEDING

5. Conclusions

We had great success with applying genetic algorithms to the problem of finding keys for substitution ciphers. We attribute this mainly to the efficiency at which genetic algorithms can search a large problem space as well as our implementation of domain specific knowledge to create a better measure of fitness for individuals.

Using the default parameters mentioned in the Experimentation section above we were almost always able to converge to solution in under a hundred generations.

However this problem is extremely sensitive to local maxima and we found through experimentation that higher mutation rates resulted in a much higher probability of convergence. We also found it beneficial to increase the percentage of elitism and believe this is mainly due to our decision to use tournament selection for deciding on which individuals can procreate.

We believe the use of genetic algorithms for this problem offers several advantages of the traditional cryptographic techniques as it no longer requires human intervention. However more work must be done on determining when to terminate the algorithm.

6. Future Works

Several possibilities exist to extend upon the work presented in this paper. Firstly the fitness function could be altered to assign fitness to individuals both based on the number of n-grams found as well as the number of actual words found increasing the probability and rate of convergence.

Furthermore the technique discussed here could be extended to deal with polyalphabetic ciphers which are ciphers where the substitution alphabet changes continuously. One example of such a cipher is the Enigma machine cipher which was used by the Germans to encrypt the communications during the Second World War.

References

[1] Brad Miller & David Goldberg, Genetic Algorithms, Tournament Selection, and the Effects of Noise, IlliGAL Report No. 95006

[2] Mat Buckland, AI Techniques for Game Programming, Premier Press 2002.